High Performance Convolutional Neural Networks for Document Processing

Kumar Chellapilla, Sidd Puri, Patrice Simard

High Performance Convolutional Neural Networks for Document Processing

Kumar Chellapilla
Microsoft Research
One Microsoft Way,
Redmond, WA, 98052, USA
kumarc@microsoft.com

Sidd Puri
Microsoft Research
One Microsoft Way,
Redmond, WA, 98052, USA
siddpuri@microsoft.com

Patrice Simard
Microsoft Research
One Microsoft Way,
Redmond, WA, 98052, USA
patrice@microsoft.com

Abstract

Convolutional neural networks (CNNs) are well known for producing state-of-the-art recognizers for document processing [1]. *However, they can be difficult to implement and are usually slower than traditional multi-layer perceptrons (MLPs). We present three novel approaches to speeding up CNNs: a) unrolling convolution, b) using BLAS (basic linear algebra subroutines), and c) using GPUs (graphic processing units). Unrolled convolution converts the processing in each convolutional layer (both forward-propagation and back-propagation) into a matrix-matrix product. The matrix-matrix product representation of CNNs makes their implementation as easy as MLPs. BLAS is used to efficiently compute matrix products on the CPU. We also present a pixel shader based GPU implementation of CNNs. Results on character recognition problems indicate that unrolled convolution with BLAS produces a dramatic 2.4X–3.0X speedup. The GPU implementation is even faster and produces a 3.1X–4.1X speedup.*

Keywords: Convolutional neural networks, BLAS, GPU.

1. Introduction

Convolutional neural networks (CNNs) are well suited for solving visual document tasks that rely on recognition and classification [1,3]. In contrast to fully connected neural networks (NNs), CNNs have been shown to be simpler to build and use. They present a flexible architecture that does not require complex methods, such as momentum, weight decay, structure-dependent learning rates, averaging layers, tangent prop, or even finely-tuning the architecture [1]. CNNs have also achieved the state-of-the-art results for character recognition on the MNIST data set of handwritten English digit images [2].

Typical multilayer-CNNs comprise layers of convolutional nodes followed by layers of fully connected nodes. For example, the best performing architecture from [1] is shown in Figure 1 and has two convolutional layers followed by two fully connected layers. The first and second convolutional layers have five and fifty nodes, respectively. Each convolutional node uses a 5×5 kernel. The kernel parameters comprise the weights that are learned during training. The hidden layer has 100 nodes and there are 10 output nodes corresponding to 10 digits.

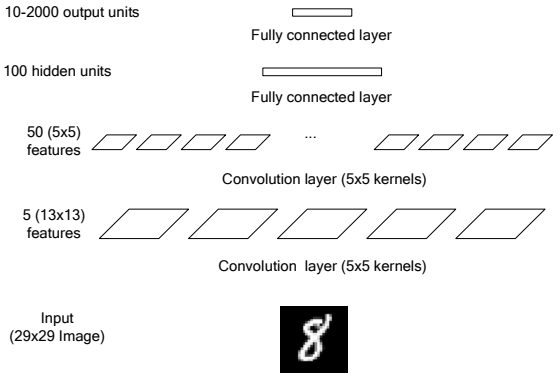

Figure 1. Convolutional Neural Network (CNN) architecture for handwritten digit recognition [1].

The weights (free parameters) in the convolutional layers are shared (see [1] for details). As a result, even though the whole CNN in Figure 1 has 133,780 weights, only 6,430 of these (less than 5%) are in the convolutional layers. Thus, from a memory and capacity standpoint the CNN is not much bigger than a regular two layered neural network. However, at runtime the convolution operations are computationally expensive and take up about 67% of the time. This makes typical CNNs about 3X slower than their fully connected equivalents (size-wise).

The computational complexity of the convolution layers stems from three sources: a) the convolution operation, b) small kernel sizes, and c) cache unfriendly memory access. Firstly, in a convolutional node, the number of operations per input grows quadratically with the length the kernel. So, in comparison with a 5×5 kernel, a 7×7 kernel needs almost twice as many operations per input pixel. Secondly, typical CNNs use kernel sizes that range from 3×3 to 9×9. Convolution operations are commonly implemented as four nested loops, the outer ones iterating over the input image (*x*- and *y*-directions) and the inner ones iterating over the kernel (*x*- and *y*-directions). Small kernel sizes make the inner loops very inefficient as they frequently incur JMP

instructions. Thirdly, the forward- and back-propagation steps require both row-wise and column-wise access to the input and kernel images. Since, 2-D images (and kernels) are commonly represented using contiguous chunks of memory in a row-wise-serialized order, column-wise access to data can result in a high rate of cache misses in the memory subsystem. This relatively slow execution of CNNs has motivated research into improving the speed of the convolutional layers in a CNN.

This weight-sharing technique has an interesting computation sharing property that can be exploited when processing large input images through sequential scanning. Classically, CNNs are shown a single character as input, and have a single set of outputs. However, CNNs can be scanned (replicated) over large input fields containing multiple unsegmented characters (whole words) very economically by simply performing the convolutions on larger inputs. Instead of producing a single output vector, they produce a series of output vectors. The outputs detect and recognize characters at different (and overlapping) locations on the input. These multiple-input, multiple-output CNN are called Space Displacement Neural Networks (SDNN) [4]. However, these speedups are accessible only when whole words or lines of text are processed in a batch.

The Graphics processing unit (GPU) is a single-chip processor that is designed to accelerate the real-time three-dimensional (3D) graphics that are displayed to a user. Initially a feature of high-end graphics workstations, the GPU has found its way onto the personal computer bus as an accelerator of graphics functions for which a conventional central processing unit (CPU) was ill-suited or simply too slow.

Current trends in GPU design and configuration have given them larger dedicated memory, higher bandwidth to graphics memory, and increased internal parallelism. In addition, current GPUs are designed with ever-increasing degrees of programmability. With the introduction of programmability, the GPU has gained enough flexibility to find use in non-graphics applications. Furthermore, the data-parallel architecture of GPUs delivers dramatic performance gains, compared to CPUs for computationally-intensive applications. Extensions to alternative graphics algorithms and scientific computing problems have been explored in a number of instances [8-9].

In this paper, we present three novel approaches to speeding up CNNs: a) unrolling convolution, b) using BLAS (basic linear algebra subroutines), and c) using GPUs (graphic processing units). It is well known that processing in the fully connected layers can be represented as matrix-vector products followed by a non-linearity. Our new unrolled convolution extends this property to convolutional layers. By unrolling, the processing in each convolutional layer becomes a matrix-matrix product. This matrix product property applies to both the forward- and back-propagation steps. BLAS is well suited for efficiently computing these matrix products. We also present a pixel shader based GPU implementation of CNNs. The GPU approach uses multiple ALU pipelines in parallel and exploits weight- and input-sharing in the convolutional layers.

2. Unrolling convolutional layers

A CNN applies a succession of convolutions to extract features (intermediate outputs from convolutional layers) that are then processed by the fully connected layers to perform high level classification. As already discussed, a straight forward implementation of convolution is not well suited for fast execution on modern computers. We address this problem by reorganizing and duplicating the inputs to the convolution in a manner that is much more regular and better suited for modern CPUs. The CPU can then take advantage of special parallel hardware (SSE or MMX) to speed up convolution substantially.

The central idea is an unfolding and duplication of the input and a rearrangement of the kernel parameters that produces a CPU friendly ordering. Using this approach each convolutional layer is converted to a matrix product during forward propagation. A nice byproduct of this is that we can simply write down the back-propagation step as another matrix product.

Simple unfolding of convolution is a well known technique. It is commonly implemented in signal processing and communications applications. For example, Matlab® has two functions, convmtx and convmtx2 (signal processing toolbox) which create “convolution matrices” in order to transform convolution into a matrix multiplication. Figure 2 presents an example depicting the traditional convolution operations in a convolutional layer. For simplicity, the bias weights, sub-sampling, and non-linearity have been omitted. The layer takes three 3×3 input features (images) and outputs two 2×2 output features (images). The inputs are convolved with their associated kernels and added together to obtain the outputs. With M inputs and N outputs, there are a total of $M \times N$ convolutions, each with its own unique kernel. In Figure 2, $M = 2$ and $N = 3$, giving a total of 6 kernels.

Figure 2 also presents an example showing the matrix product version of the convolutional layer. The input for each convolution is rewritten such that each row contains all the input values necessary to compute one element of an output feature. This implies duplication of some inputs. For example, the center of each 3 by 3 input feature is used 4 times to compute each element of an output feature. Instead of accessing the center input 4 times it is simply duplicated 4 times through copying. The process of copying the input value is done once, regardless of the number of output features. This means that if there are 50 output features, the cost of copying will be negligible compare to the cost of computing all the output features. With multiple input features, since the results of the convolutions are summed across input features, the input features can be concatenated into one

long row of the input matrix. The kernels are also unrolled and concatenated to produce the kernel matrix. The *i*-th column contains all kernels that produce the *i*-th output. Each output feature then corresponds to a column in the new kernel matrix. When the new input matrix is multiplied by the kernel matrix, the output features are computed automatically. They can then be rolled back into 2D matrices to be equivalent to the original 2D layout. For efficiency, the kernels can be stored directly as the kernel matrix, rather than a stack of 2D images. We note that the cost of rolling and unrolling operations for the inputs and outputs are negligible compared to the number of operations in the matrix product.

Figure 3 presents the general version of the unrolled convolution layer with O_f nodes. It takes $I_x \times I_y$ input features, has $K_x \times K_y$ sized kernels, sub-samples convolution outputs by S_x and S_y along the *x*-, and *y*-directions, respectively, and produces $O_x \times O_y$ sized outputs. Unrolling produces input (X), kernel (W), and output (Y) matrices that capture the convolution operations as matrix products. The forward-propagation, backward-propagation, and weight updates take the following familiar forms:

Forward-prop: $Y = X * W$ (1)

Back-prop: $\nabla_X = \nabla_Y * W^{\mathrm{T}}$ (2)

Weight-Gradient: $\nabla_W = X^{\mathrm{T}} * \nabla_Y$ (3)

where ∇_X , ∇_Y , and ∇_W are the input, output, and weight gradients respectively. The weight update is then accomplished using

Weight-Update: $W_{new} = W_{old} - \eta \nabla_W$ (4)

where η is the learning rate.

3. Basic linear algebra subroutines (BLAS)

Unrolling convolution solves the small kernel size problem and pools small convolutions into a large matrix-matrix product. The result is also a notationally simple and clear description of the processing steps in the convolution layers. However, naïve implementations of matrix products can be slow compared to what is achievable on today's computers.

Efficiently computing large matrix-vector and matrix-matrix products is a well studied problem. Basic Linear Algebra Subroutines (BLAS) contains subprograms for basic operations on vectors and matrices. BLAS was designed to be used as a building block in other codes such as LAPACK (Linear Algebra Package). The source code for one implementation of BLAS is available from Netlib. However, many computer chip manufacturers provide special versions of BLAS tuned for maximal speed and efficiency on their chips [5-6]. Automatically tunable versions of BLAS such as ATLAS [7] are also commonly available.

One of the main advantages of BLAS is that the calling sequences are standardized so that programs that call BLAS will work on any computer that has BLAS installed. If you have a fast version of BLAS, you will also get high performance on all programs that call BLAS. Hence BLAS provides a simple and portable way to achieve high performance for linear algebra computations.

4. Convolution using the Graphic Processing Unit (GPU)

Our work on using GPUs is motivated by the success of the GPU implementation of a two layered fully connected neural network presented in [9]. We propose a GPU implementation of convolutional neural networks both for training and testing. The first basic advantage that GPUs have over CPUs is their ability to access memory in a 2-D manner, i.e., row access and column access to memory take the same amount of time, even with very large matrices containing several thousand rows and columns. The second advantage is that GPUs contain several parallel pipelines with independent arithmetic logic units (ALUs) that can significantly speedup computations. It is not uncommon to find ALUs that have 16, 24, or even 48 pipelines. However, GPUs do not have a memory cache and do not offer primitive operations for looping and branching. Implementing equations (1)-(4) using SIMD instructions and no explicit looping and conditionals is a challenging task.

4.1. GPU Setup

Following the common practice of general-purpose GPU programming, our implementation is written as a series of "pixel shaders." We used Pixel Shader model 3.0 in our implementation. To set this up, we create a scene consisting of a single triangle that covers the entire viewport (see [9] for details). This allows us to ignore all of the stages of the graphics processor other than the pixel shader. Each of our data-parallel variables is stored as a two-dimensional "texture." Some variables have more than two dimensions, so they need to be embedded into a two-dimensional texture. For example, we "flatten" the kernel parameters in a manner similar to that in Figure 3.

4.2. GPU Passes

Each invocation of a pixel shader is referred to as a "pass." At each pass that we perform, there is exactly one output texture. The pixel shader program is invoked once at each pixel of the output texture, and the result of executing the shader is stored in the output texture at that pixel. During the execution of the shader, we can read values from up to 16 input textures[1], but not from the output texture.

4.3. Reducing the number of passes

Equations (1)-(4) are decomposed into a sequence of GPU passes. It is not uncommon to split up each of the equations onto several passes. In our implementation, the

[1] The number of input textures is a function of the graphics card used and has been steadily increasing with with each new version.

number of passes varied from 2-4 per layer for the forward pass and 5-6 per layer during the backward pass. Error computation is done using two passes. Overall, the computation for one fwd-prop and one back-prop was distributed over 37 passes.

Each pass carries with it a fixed performance overhead. This overhead is in addition to the time needed to perform the actual computations during the pass. In addition, the compiler contained in the GPU driver has no way of optimizing across pixel shaders. Both of these factors make it important to coalesce as much computation as possible in each pixel shader in order to achieve good performance.

4.4. Reduce operations: Summations and Matrix-Vector products

The data-parallel nature of pixel shaders and the lack of explicit loops makes it hard to perform reduce operations such as summations and matrix-vector products. Summation is achieved over several passes, where each pass sums some fixed number, say r, of horizontally adjacent patches. Using this approach, the number of passes required to sequentially compute the sum is $\log_r N$.

A matrix-vector product contains an implicit sum. To compute it, we first compute the body of the summation, which is effectively a matrix. Then we use the above algorithm to reduce the intermediate result into a vector. The entire procedure requires $\log_r (N+1)$ passes. We achieved the highest performance with $r = 10$.

4.5. Allocation and Communication

Allocation and deallocation on the GPU is an expensive operation. To minimize costs, we define all our pixel shaders and allocate all our textures at the beginning of our program, and reuse them until we exit.

Transmission of data from the CPU to the GPU and vice versa is usually considered expensive [9], but in our case the amount of data was small and there was no need to optimize communication overhead. For each training sample, we need to transmit only the pixel patch corresponding to the input (usually 29×29 pixels) and the correct classification.

4.6. Conditionals

Because of the SIMD nature of the GPU, conditionals can be a significant burden on performance. For example, current implementations of conditionals on GPUs evaluate both branches which is wasteful. In pixel shader code, there are several different ways to encode these conditionals. We tried the `if` instruction, the `cmp` instruction and the (`p0`) predicate, and found each of them to be slow.

In the case that the conditional checks for edge conditions, the fastest solution is to elide the conditional altogether, but set the input stage of the GPU to "border color sampling." This allows for all accesses outside of the defined area of a texture to return 0. For non-border conditionals, we found it most efficient to encode the condition into a floating point number which is greater than 0 iff the condition is true. This number can be normalized to be exactly 0 or 1 by using the `mul_sat` instruction with a sufficiently large factor.

5. Experiments

Experiments were conducted using two character recognition problems with 10 (MNIST) and 94 (Latin) classes, respectively. The MNIST dataset [2] consists of 60,000 hand written digits uniformly distributed over 0-9. The Latin character set consists of 94 most common ASCII characters used in the English language.

Three different input image sizes were used: 29x29, 37x37, and 61x61. BLAS experiments were conducted on an Intel Pentium 4 (2.8GHz) using BLAS routines from the Math Kernel Library® (ver. 7.2) from Intel. GPU experiments used the NVIDIA GeForce 7800 Ultra graphics card[1] with DirectX 9.0c and Pixel Shader 3.0. The unrolled, BLAS, and GPU implementations were tested for accuracy and speed against a conventional C++ implementation from [1].

6. Results

All three implementations were tested to ensure correctness (both during training and testing) against the C++ implementation. Since the GPUs do not support full IEEE floating point conformance, we relied on training rates and final solution quality. Each version successfully trained convolutional neural networks that achieved test error rates that were less than 1.2% [2].

As expected, the rate of training as a function of the number of epochs is nearly the same between the three implementations. However, wall clock training times varied dramatically. Table 1 presents the timing and speedup results for CPU without BLAS, CPU with BLAS, and GPU. Both CPU implementations used unrolled convolution. CPU times without unrolled convolution are slightly longer (both with and without BLAS) than those for CPU without BLAS but with unrolled convolution. As a result, these have been omitted from Table 1. In other words, unrolling convolution did not give a speedup when BLAS was not used. Further, with the Intel MKL BLAS implementation we found no speedups with vectors that had less than 65 floats. For matrices, this threshold was a row/column length of 65 floats.

The provided numbers are wall clock times for a batch of 1000 forward-props and 1000 backward-props. For both problems and all input sizes using unrolled convolution with BLAS gave a dramatic speedup that ranged from 2.42X – 3.00X. The GPU implementation

[1] NVIDIA GeForce 7800 Ultra has 24 pipelines. We are also working on obtaining results on the new ATI's Radeon X1900 that has 48 shader units. We hope to have these results in the next few weeks and will include them in the paper.

Table 1. Times and Speedups for 100 fwd-props and 100 back-props for the three different implementations of convolutional neural networks: CPU without BLAS, CPU with BLAS, and GPU. Both CPU implementations used unrolled convolution.

		Time in secs for 1000 fwd-props & 1000 back-props									Speed up						Speed up Ratio		
	Architecture	CPU no BLAS			CPU with BLAS			GPU			BLAS speedup			GPU speedup			GPU / BLAS		
	Input size =>	29	37	61	29	37	61	29	37	61	29	37	61	29	37	61	29	37	61
MNIST	5, 50,100,10	6.28	11.43	37.74	2.58	4.57	14.49	2.02	3.27	10.00	2.43	2.50	2.60	3.11	3.49	3.77	1.28	1.40	1.45
	5, 50,250,10	10.86	19.81	66.43	4.05	7.30	23.75	3.12	5.66	18.80	2.68	2.71	2.80	3.48	3.50	3.53	1.30	1.29	1.26
	5,100,100,10	12.28	21.15	70.93	4.54	8.07	25.27	3.14	5.63	18.60	2.70	2.62	2.81	3.91	3.75	3.81	1.44	1.43	1.36
	5,100,250,10	20.65	37.68	135.57	7.34	13.47	45.20	5.35	10.32	36.41	2.82	2.80	3.00	3.86	3.65	3.72	1.37	1.31	1.24
	10, 50,100,10	9.00	15.81	47.84	3.53	6.20	19.60	2.52	4.21	12.97	2.55	2.55	2.44	3.57	3.76	3.69	1.40	1.47	1.51
	10, 50,250,10	13.45	24.09	77.73	5.00	8.89	28.81	3.63	6.62	21.83	2.69	2.71	2.70	3.70	3.64	3.56	1.38	1.34	1.32
	10,100,100,10	16.97	28.95	91.64	5.98	10.54	34.55	4.13	7.44	24.18	2.84	2.75	2.65	4.11	3.89	3.79	1.45	1.42	1.43
	10,100,250,10	25.05	45.37	154.82	8.78	16.00	53.40	6.35	12.12	41.91	2.85	2.84	2.90	3.95	3.74	3.69	1.38	1.32	1.27
LATIN	5, 50,100,94	6.70	11.55	36.49	2.73	4.70	14.49	2.20	3.54	10.44	2.45	2.46	2.52	3.04	3.26	3.49	1.24	1.33	1.39
	5, 50,250,94	11.18	20.02	66.43	4.28	7.52	24.02	3.22	5.76	19.00	2.61	2.66	2.77	3.47	3.47	3.50	1.33	1.30	1.26
	5,100,100,94	12.34	21.29	71.15	4.61	8.17	26.39	3.38	5.94	19.06	2.67	2.61	2.70	3.65	3.59	3.73	1.36	1.38	1.38
	5,100,250,94	20.60	37.78	134.95	7.53	13.66	45.35	5.45	10.42	36.51	2.74	2.77	2.98	3.78	3.62	3.70	1.38	1.31	1.24
	10, 50,100,94	9.09	15.66	48.02	3.62	6.29	19.69	2.72	4.50	13.42	2.51	2.49	2.44	3.34	3.48	3.58	1.33	1.40	1.47
	10, 50,250,94	13.76	24.24	77.95	5.20	9.19	29.12	3.73	6.73	21.94	2.65	2.64	2.68	3.68	3.60	3.55	1.39	1.37	1.33
	10,100,100,94	17.34	28.56	91.00	6.19	10.75	34.59	4.38	7.74	24.56	2.80	2.66	2.63	3.96	3.69	3.71	1.41	1.39	1.41
	10,100,250,94	25.23	45.05	154.23	9.00	16.19	53.67	6.46	12.23	42.02	2.80	2.78	2.87	3.91	3.68	3.67	1.39	1.32	1.28

was even faster with speed gains ranging from 3.11X – 4.11X. On average the GPU implementation was 24% – 47% faster than the CPU with unrolled convolution and BLAS. It is clear that the speedups are quite consistent over different network sizes and between the two implementations.

For both implementations, faster speedups are observed on larger network sizes. As networks get very large the GPU implementation will likely scale better that the CPU implementation. The CPU implementation will scale well till the neural network memory footprint becomes comparable with the on chip cache size. Exceeding the cache size will adversely affect performance and limit further gains. On the contrary, the GPU does not suffer from this problem as the pixel shader pipelines do not use a cache.

7. Conclusion

We presented three independent approaches for high performance implementation of convolutional neural networks for document processing using today's CPUs and GPUs. Experimental results on real-world recognition problems show that dramatic speedups of up 3.00X on the CPU and 4.11X on the GPU. Given the current trends towards multi-core CPUs and GPUs with numerous pipelines, these speedups are only expected to increase in the future.

References

[1] P. Y. Simard, D. Steinkraus, & J. Platt, "Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis," *International Conference on Document Analysis and Recognition (ICDAR),* IEEE Computer Society, Los Alamitos, 2003, pp. 958-962.

[2] The MNIST database of handwritten digits http://yann.lecun.com/exdb/mnist/

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[4] O. Matan, C.J.C Burges, Y. LeCun and J. S Denker (1992), "Multi-Digit Recognition Using a Space Displacement Neural Network," in *NIPS*'92.

[5] Intel Math Kernel Library available from http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm

[6] The AMD Core Math Library (ACML) available from http://developer.amd.com/acml.aspx

[7] Automatically Tuned Linear Algebra Software (ATLAS) available from http://math-atlas.sourceforge.net/

[8] J. Kruger, and R. Westermann, "Linear Operators for GPU Implementation of Numerical Algorithms," *Proceedings of SIGGRAPH*, San Diego, 2003, pp. 908-916.

[9] D. Steinkraus, I. Buck, and P. Y. Simard (2005), "GPUs for Machine Learning Algorithms," *ICDAR 2005*, vol. 2, pp. 1115-1120.

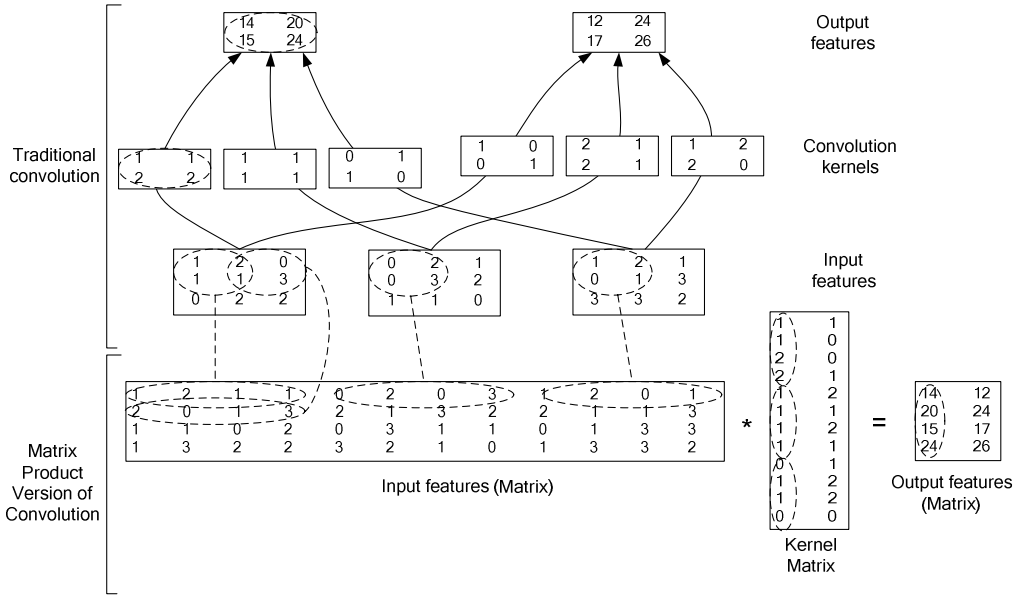

Figure 2. Example convolution operations in a convolutional layer (biases, sub-sampling, and non-linearity omitted). The top figure presents the traditional convolution operations, while the bottom figure presents the matrix version.

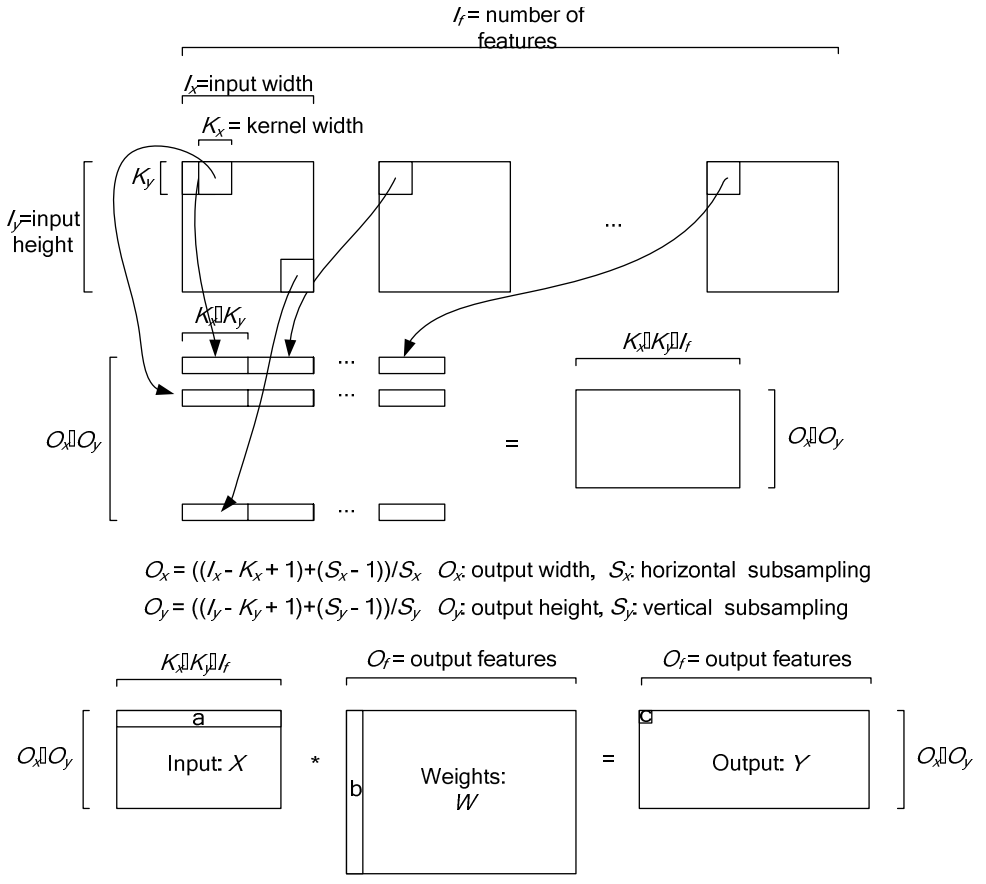

Figure 3. Unrolling the convolution operations in a convolutional layer (biases, sub-sampling, and non-linearity omitted), to produce a matrix-matrix product version.